Applying Streaming SIMD Extensions to Collision Detection

Version 1.1

01/99

Order Number: 243646-002

Table of Contents

Revision History

Revision	Revision History	Date
1.1	FCS revision.	01/99

1 Introduction

Streaming SIMD Extensions for the Intel® Architecture (IA) instruction set provide floating-point single-instruction, multiple-data (SIMD) instructions. These instructions provide a means to accelerate operations typical of 3D graphics, real-time physics, and spatial (3D) audio. This application note discusses the topic of collision detection as applied to 3D geometry applications.

Streaming SIMD Extensions can increase collision detection computation performance by about three times over standard x87 code. For this study, the Oriented Bounding-Box (OBB) algorithm was selected, a simulator was constructed to exhaustively analyze OBB, three core computational regions were identified and coded with standard x87 and Streaming SIMD Extensions, and conclusions were drawn from the benchmarked results.

This paper first presents a background for the selection of a collision detection algorithm, then discusses a specific implementation using the Streaming SIMD Extensions.

2 Collision Detection with Streaming SIMD Extensions

The process of projecting the gains possible with Streaming SIMD Extensions in the area of collision detection was divided into three phases:

1) Selection of a representative algorithm
2) Determination of the critical functions
3) Simulation and benchmarking of the critical functions

2.1 Selection of a Representative Algorithm

A number of algorithms were evaluated for likelihood of use, both for the present and the future; these included the Axis Aligned Bounding-Boxes (AABB), Bounding-Spheres, Closest Features, and Oriented Bounding-Boxes (OBB) algorithms. There are a large number of collision detection algorithms. However, most of them (such as AABB, Bounding-Spheres, and OBB) share a common core design element: break the environment up into a hierarchical representation and recursively test nodes for intersection.

Selection of an algorithm focused on this “class” of algorithms because of the somewhat ubiquitous acceptance of this central foundation to collision detection. Of the many algorithms of this type, OBB was selected for intensive study for the following reasons:

- It has exceptional performance capabilities over a wide range of environmental conditions
- It is widely used in the industry

For more information, refer to Section 3 for a comprehensive discussion of the Oriented Bounding Box algorithm. In addition, a brief study was performed of the Bounding Spheres method; this information is presented in Section 6.

2.2 Determination of the Critical Functions

The OBB core function basically compares two bounding-boxes for intersection. This function consists of fifteen conditional equations. Any single condition that evaluates “true” indicates that the two bounding-boxes do **not** intersect. These fifteen conditions are divided into three classes of similarly structured equations (differing only in coefficients), with three in Class I, three in Class II, and nine in Class III (refer to Section 3.3). Only if all three classes of equations fail do the two bounding-boxes actually intersect.

Since non-intersection can be determined by any one of the fifteen conditions, it follows that evaluating all fifteen equations every time is unnecessary, and that performance could be improved by adjusting the evaluation order. As a result, a "universe simulator" was constructed to determine the probability that a given condition would detect non-intersection.

The universe simulator accepts as arguments the boundary of the universe and the ratio of bounding-box sizes. Its method is to randomly orient the bounding-boxes, and then test for intersection over hundreds of thousands of iterations. This type of simulation made it possible to understand the distribution of the conditions that "catch" the most non-intersections. Tuning the algorithm’s performance was then just a matter of selecting the cases that catch the most non-intersections, and evaluating these first.

This selective computation allows non-intersecting bounding-boxes to be identified with the minimum computation. Through a range of best and worst case universes, a combination of statistical analysis, and ease-of-coding determined that to minimize branching and unnecessary calculation, non-intersection should be tested in three separate places in the function:

- After the first three conditional equations
- After the first six conditional equations
- After all fifteen conditional equations

This means that all equations in an entire class of conditions are evaluated in series (or simultaneously with Streaming SIMD Extensions), with a single branch that returns from the function if any of the class's conditions evaluated to be true. For example, the first three equations (all of Class I) are evaluated in their entirety before any condition is tested to see if it is true. Even if the first equation catches non-intersection, the second and third are still evaluated before the non-intersection is detected. While this method does not altogether eliminate the evaluation of unnecessary conditional equations, the waste was determined to be minimal in comparison to the penalties associated with additional branching.

2.3 Simulation and Benchmarking of the Critical Functions

Based on this general function flow decision, a baseline version was built using only C code. To reduce the code size and cost per instruction (CPI) of the C version, the first class of conditional equations was re-coded in x87 assembly. The resulting performance gain (approximately 1.3x) was projected onto the remaining two groups, from which it was possible to estimate how an entirely x87 baseline version would perform.

Building the Streaming SIMD Extensions version required an analysis of the conditional equations to find areas where calculations could be done in parallel. Due to the large amount of data-reuse in similar equations, it was decided that portions of each of the three classes could be calculated in parallel. This type of evaluation using the Streaming SIMD Extensions and registers is illustrated in Figure 1.

Each class contains similarly structured equations that can be performed in parallel. Up to four conditional equations can be processed simultaneously with SIMD instructions. There are three Class I equations, so this entire class can be evaluated simultaneously using three of the four operands. The same applies to Class II. There are nine Class III equations, where a maximum of four can be processed at a time. Regardless of the divisions chosen, the set of nine Class III equations must be split into at least three groups for computation. Computing three groups of three equations allows greater register reuse than computing two groups of four equations and one group of one equation.

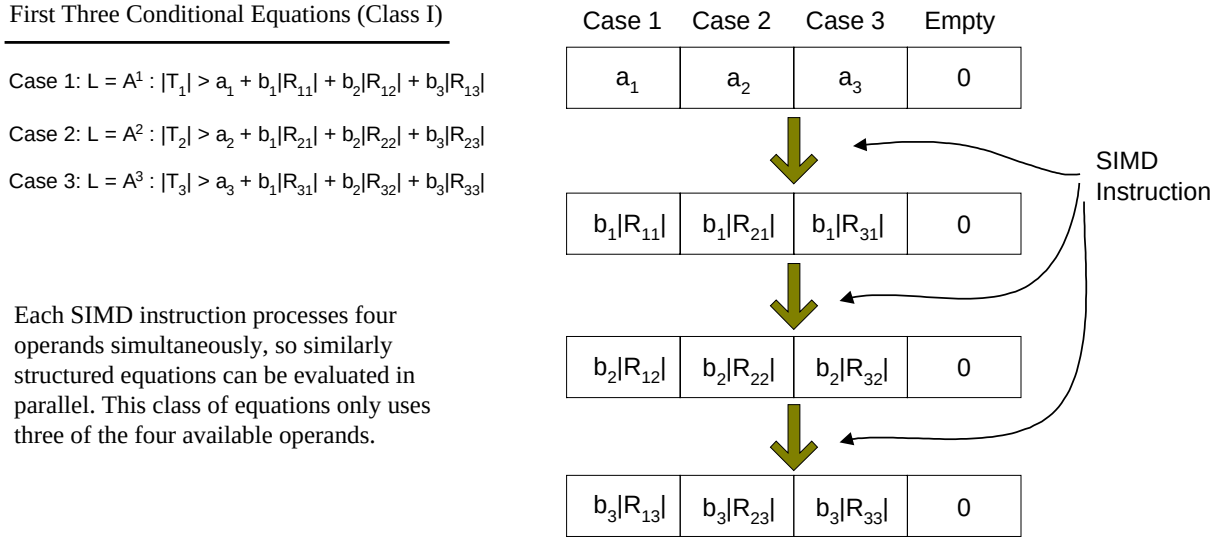
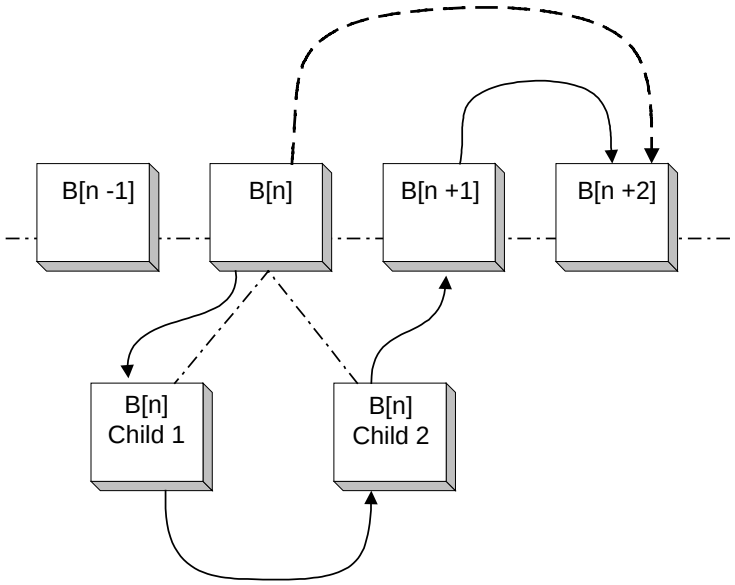


Figure 1: Parallel evaluation of conditional equations using SIMD instructions

Next, it was necessary to choose an appropriate prefetching scheme to match the data-access pattern of the algorithm. In a traditional geometry-transformation pipeline, Data[n+1] is processed immediately after Data[n]. Because of this linear data-access pattern, choosing the data to prefetch is obvious. Conditional hierarchical collision detection is much different because it isn't known what data will be required in the near future.

Normally, for a particular iteration, if it is determined that bounding-boxes A and B[n] collide, the child bounding-boxes of A and B[n] are tested for intersection starting with the next iteration. As a result, this is much different than prefetching for a linear graphics pipeline because data structure B[n+1] isn't necessarily processed immediately after B[n]. Depending on how often intersections occur and the organization of the data structures in memory, it is possible that a seemingly random memory access pattern could result.

To achieve the benefits of prefetching, the assumption was made that in an application using collision detection, a single bounding box (A) often would be compared to a large list of bounding boxes (B[#objects]), such as comparing a player's bounding-box to the bounding-boxes of the furniture in a room. Since all bounding-boxes in this array require testing at some point, then regardless of the result of the current iteration it is known that the data for bounding-boxes of future iterations will be used eventually. For a particular iteration, the function is calculating the intersection between box A and box B[n], and will begin prefetching data for box B[n+3]. This gives the processor at least three iterations of the function to load the required two cache-lines of data into the cache-hierarchy. In addition, this study assumed that there was a type of stack-based Test Queuing system in place above the actual intersection function, allowing for linear prefetching characteristics (this is illustrated in Figure 2). For more information about Test Queuing and several other possible prefetching schemes, refer to Section 4.



When testing for a collision between bounding box A and B[n], the function prefetches bounding box B[n+2] (the dashed line). If that assumption is wrong and a collision is detected, one possibility is to test the children of B[n] before B[n+1] (the testing path is is shown by the solid line). Regardless, box B[n+2] is eventually tested for collision, and its data could be available without a data-stall because it has been prefetched.

Figure 2: Simple linear progression prefetching

3 The Oriented Bounding-Box Collision Detection Algorithm

Of the multitude of collision detection algorithms available, many rely upon breaking a model into a binary tree of bounding-volumes, the leaf nodes of which correspond to individual polygons. The Oriented Bounding-Box (OBB) algorithm is no different.

The three fundamental parts to the Oriented Bounding Box algorithm are:

- Organizing the bounding box hierarchy
- Building a bounding box
- Detecting the collisions

3.1 Organizing the Hierarchy

The purpose of organizing bounding-boxes into a hierarchy is to quickly reduce the set of bounding boxes that are potentially intersecting the bounding-box in question. For instance, assume that it is desired to know if a player in a video game is colliding with any objects in a house. First a test would be done to see if the player is actually inside of the house. If so, then it would be seen what floor the player is on, followed by which room and so on. At the end of this hierarchy of detail would be individual objects, or possibly even individual polygons (to see where the player is touching on a single object). The actual building and balancing of a hierarchical tree of bounding-boxes for the shortest search path (and other necessary consideration) is outside of the focus of this study.

3.2 Building a Bounding-Box

Basically a "tight fitting" bounding-box is computed using a complicated technique (also not the focus of this study) that aligns the axes of the bounding box along the natural axes of the group of vertices being bounded. Unlike the Axis-Aligned Bounding-Box (AABB) and Bounding-Spheres methods of

collision detection, very little extraneous space is contained within the bounding-boxes of the tree, thereby minimizing unnecessary intersection tests.

Figure 3 shows a representation of how bounding boxes can be computed. Part (a) of the diagram shows an object to be divided into parent and child bounding volumes. Part (b) shows the division of parent and child using Oriented Bounding Boxes (OBB), Part (c) shows the division using Bounding Spheres, and Part (d) shows the division using Axis-Aligned Bounding Boxes (AABB). For the examples shown in Parts (b), (c), and (d), the hatched line volume is the parent bounding volume, and the solid line volumes are the children.

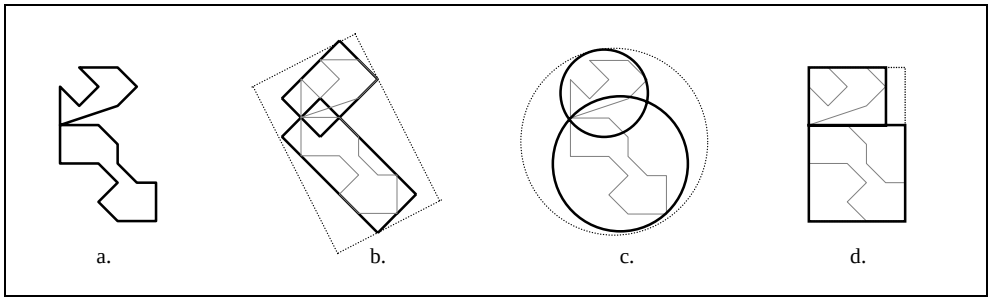


Figure 3: Construction of bounding boxes with different collision detection methods

3.3 Detecting Collisions

All collisions occur between two bounding-boxes, named A and B. Each bounding-box is centered upon its respective coordinate system origin with axes $\mathbf{A}^i$ or $\mathbf{B}^i$, as well as variables a_i or b_i associated with the length that the box extends along its own axes, for i=1,2,3. Box B is transformed into the coordinate system of box A by a translation vector **T** and a 3x3 rotation matrix **R**, with the ith column of **R** being synonymous with the vector $\mathbf{B}^i$ (assuming column-vector representation). Refer to Figure 4.

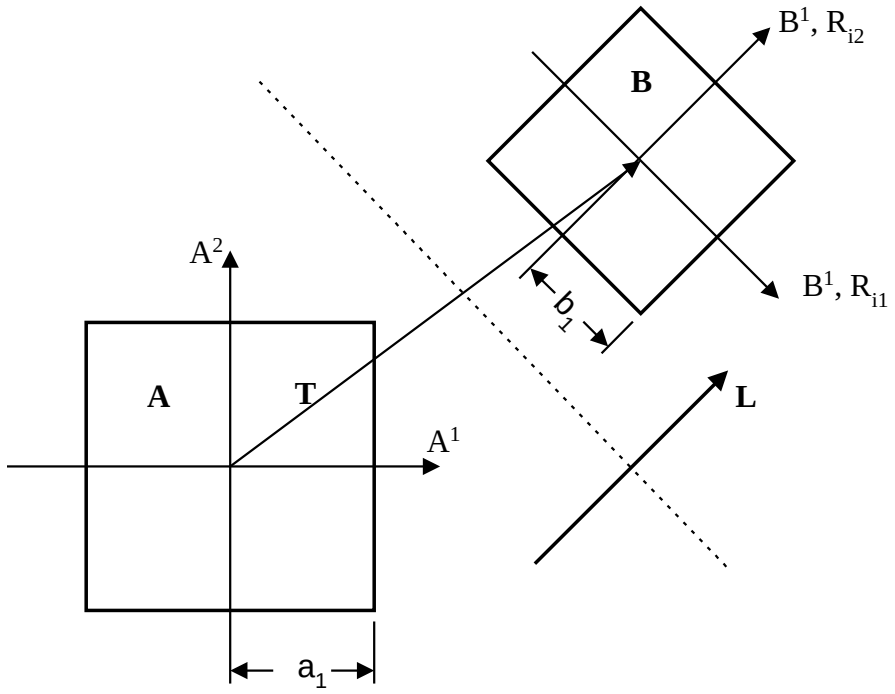


Figure 4: Variables in the intersection test

Testing for intersection consists of searching for the existence of a unit vector **L** that the magnitude of the sum of the projections of the boxes' axes onto **L** is less than the magnitude of **T** projected onto **L**. This amounts to trying to find the normal to a plane which wholly contains A and B on opposite sides. If this vector **L** exists, **L** is called a "separating axis" and boxes A and B are said to be "disjoint", or non-intersecting.

While L can be any vector in space, it can be shown that if A and B are disjoint, evaluating the set of fifteen possible combinations of box A's and box B's axes will find a separating axis. In addition, when **L** is chosen to be a combination of $\mathbf{A}^i$ and $\mathbf{B}^i$, many simplifications can be found to make the evaluation simpler. As a result, the fifteen conditional equations come down to those mtshown in Figure 5:

Class I

Case 1 : $L=A^1 : |T_1| > a_1 + b_1|R_{11}| + b_2|R_{12}| + b_3|R_{13}|$

Case 2 : $L=A^2 : |T_2| > a_2 + b_1|R_{21}| + b_2|R_{22}| + b_3|R_{23}|$

Case 3 : $L=A^3 : |T_3| > a_3 + b_1|R_{31}| + b_2|R_{32}| + b_3|R_{33}|$

Class II

Case 4 : $L=B^1 : |T_1R_{11} + T_2R_{21} + T_3R_{31}| > a_1|R_{11}| + a_2|R_{12}| + a_3|R_{13}| + b_1$

Case 5 : $L=B^2 : |T_1R_{12} + T_2R_{22} + T_3R_{32}| > a_1|R_{21}| + a_2|R_{22}| + a_3|R_{23}| + b_2$

Case 6 : $L=B^3 : |T_1R_{13} + T_2R_{23} + T_3R_{33}| > a_1|R_{31}| + a_2|R_{32}| + a_3|R_{33}| + b_3$

Class III

Case 7 : $L=A^1xB^1 : |T_3R_{21} - T_2R_{31}| > a_2|R_{31}| + a_3|R_{21}| + b_2|R_{13}| + b_3|R_{12}|$

Case 8 : $L=A^1xB^2 : |T_3R_{22} - T_2R_{32}| > a_2|R_{32}| + a_3|R_{22}| + b_1|R_{13}| + b_3|R_{11}|$

Case 9 : $L=A^1xB^3 : |T_3R_{23} - T_2R_{33}| > a_2|R_{33}| + a_3|R_{23}| + b_1|R_{12}| + b_2|R_{11}|$

Case 10 : $L=A^2xB^1 : |T_1R_{31} - T_3R_{11}| > a_1|R_{31}| + a_3|R_{11}| + b_2|R_{23}| + b_3|R_{22}|$

Case 11 : $L=A^2xB^2 : |T_1R_{32} - T_3R_{12}| > a_1|R_{32}| + a_3|R_{12}| + b_1|R_{23}| + b_3|R_{21}|$

Case 12 : $L=A^2xB^3 : |T_1R_{33} - T_3R_{13}| > a_1|R_{33}| + a_3|R_{13}| + b_1|R_{22}| + b_2|R_{21}|$

Case 13 : $L=A^3xB^1 : |T_2R_{11} - T_1R_{21}| > a_1|R_{21}| + a_2|R_{11}| + b_2|R_{33}| + b_3|R_{32}|$

Case 14 : $L=A^3xB^2 : |T_2R_{12} - T_1R_{22}| > a_1|R_{22}| + a_2|R_{12}| + b_1|R_{33}| + b_3|R_{31}|$

Case 15 : $L=A^3xB^3 : |T_2R_{13} - T_1R_{23}| > a_1|R_{23}| + a_2|R_{13}| + b_1|R_{32}| + b_2|R_{31}|$

Figure 5: The fifteen conditions for non-intersection

In these equations, $\mathbf{A}^i$ and $\mathbf{B}^i$ are the boxes' respective coordinate axes, $\mathbf{T}_i$ is the component of the translation vector, and $\mathbf{R}_{ij}$ is the component of the rotation matrix. If any single conditional equation evaluates to be true, the two bounding-boxes are disjoint.

4 Prefetching Schemes

Whenever a program attempts to make predictions about the future, there is the possibility of error. Prefetching falls into this category: the program is attempting to guess what data will be needed in the near future in order to avoid data-stalls. Prefetching can be performed in a number of different ways, each having its own advantages and disadvantages.

This section presents four different prefetching methods:

- Linear progression
- Parent/child progression
- Linear, parent/child combination
- Using history to direct prefetches

4.1 Linear Progression

The most straightforward method of prefetching is to assume that `B[n+k]` eventually will be needed (where k>1) when `B[n]` is being tested (see Figure 6). This method was used in part for this study and described in detail in Section 2.3.

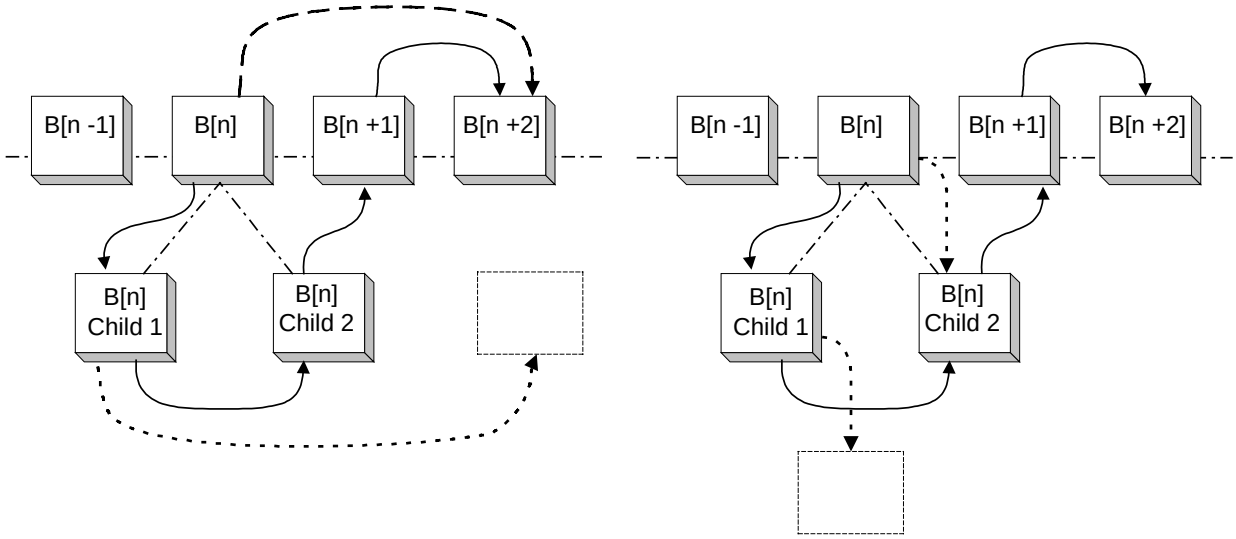


Figure 6: Linear progression vs. Parent/Child progression

4.1.1 Pros

The linear progression method is extremely simple to implement, has a somewhat high degree of accuracy, and provides moderate performance benefits over non-prefetched versions of the same function. By comparing bounding-box `A` to a straight list of bounding-boxes, `Intersect(A,B[n])` is always directly followed by `Intersect(A,B[n+1])`. This method, as previously discussed, always has the data for `B[n]` prefetched (for n>2).

4.1.2 Cons

From a higher level view, collision detection doesn't really occur between a single bounding-box and a list of bounding-boxes, but between a single *object* and a list of objects. Each object is represented as a hierarchy of bounding-boxes, where each node can have two children. As a result, if `Intersect(A,B[n])` turns out to be true, the next test isn't `Intersect(A,B[n+1])` but in fact `Intersect(A.Child1,B[n].Child1)`[1] – not a linear progression at all (see Figure 6). `B[n].Child1` was never prefetched, so this will produce a data stall. Plus, assuming:

```
B[n+k] = B[n]+sizeof(BOXDATA)*k
```

Then the test upon `B[n].Child1` will be prefetching:

```
B[n].Child1+sizeof(BOXDATA)*k
```

Which, depending on the memory organization, could be just about any bounding-box, or possibly no bounding-box at all. As a result, assuming linear progression works fine when processing a list of bounding-boxes in contiguous memory, but completely breaks down when traversing a non-linear structure, such as a binary tree.

4.2 Parent/Child Progression

To perform better prefetching for binary tree progression, prefetching a child bounding-box of the current bounding-box (*i.e.*, prefetch B[n].Child2 instead of B[n+1]) provides more immediate usefulness. As for which child to prefetch, it would be better to prefetch the second child, because it takes longer than a single iteration of the intersection function to prefetch three cache-lines of data (which is required for evaluating Class I, II and III conditions).

4.2.1 Pros

Assuming that `Intersect(A,B[n])` is true, prefetching for `B[n].Child2` begins and `Intersect(A.Child1,B[n].Child1)` is the next test. Regardless of the result, eventually `B[n].Child2` is used (just as Linear Progression was assured that eventually all of the data in the list would be used, Trickle-Down Progression assures that **as long as an intersection on `B[n]` occurs**, both `B[n].Child1` and `B[n].Child2` are used). In addition, this is still rather trivial to implement.

4.2.2 Cons

The main disadvantage to Parent/Child progression is that if an intersection does not occur, the result is a data stall for `B[n+k]` because the function assumes that an intersection will occur and prefetches `B[n].Child2` instead of `B[n+k]`. Likewise, eventually a test will either fail or not have any more children. Despite `B[n]` lacking lower-levels of data, the function attempts to prefetch what it thinks is `B[n].Child2` (a potentially random location in memory). As a result, this method is guaranteed to

[1] Because the entire bounding-box hierarchy for `A` is repeatedly used, this study assumes that `A` is always in the L1 cache and never needs to be prefetched.

prefetch at least three cache-lines of data that will not be used for each detected intersection (potentially very costly due to the exponential growth of recursively testing a binary tree).

4.3 Linear, Parent/Child Combination

As in most situations, a combination of multiple methods provides better results than a single method alone. One possible coordination of the two would be to use Linear Progression for the root nodes of the bounding-box hierarchy, while using Parent/Child Progression when an intersection is detected.

4.3.1 Pros

Using this combination of methods has the advantage of assuming that objects in the universe do not intersect, and prefetching an object ahead in the list accordingly. Likewise, if an intersection does occur, it assumes that child bounding-boxes are likely to occur as well, and will prefetch appropriately.

4.3.2 Cons

The combination of methods is more difficult to implement because it requires programming two separate intersection functions. Also, this method still suffers from the same drawbacks as the Trickle-Down Progression.

4.4 Using History to Direct Prefetches

At any particular iteration there are four possible outcomes: descending the children of A, descending the children of B, finding that A and B are disjoint, or directly testing the associated triangles. Some information is inherently available on the probability of the outcome. For instance, if one of the bounding-boxes has children, it is impossible for a triangle-triangle test to occur as both nodes are not leaves. With this and other information, it is possible to determine combinations of “hints” at each iteration. Data can be recorded from running various scenes to determine what is the most likely outcome from each of the combinations of hints, and then hard-coded into the core function. Due to the impossibility of many combinations, it may be relatively easy to detect which situation was currently in use and predict appropriately. This has a potential to bring the prediction success ratio higher.

This direction is being investigated further, and a subsequent update of this paper will contain additional information.

5 Bounding Spheres Algorithm using Streaming SIMD Extensions

While this study focused on the performance gains with respect to the Oriented Bounding-Box algorithm, there is also the potential that programmers will choose some sort of bounding-sphere algorithm for collision detection. As a result, a small performance analysis of a type of bounding-sphere algorithm was done.

5.1 Memory Requirements

Fetching data from memory is one of the most costly steps in a processing pipeline; therefore, it is beneficial to fit the required data into a whole number of cache lines (each of 32 bytes). With this in mind, it is possible to fit the required data for three whole bounding-spheres into two cache lines. Each bounding-sphere requires 12 bytes for translation information as well as 4 bytes for radius information. For three bounding-spheres this amounts to 48 bytes of data. The remaining 16 bytes of data are required to represent the relationship of these bounding-boxes to the rest of the hierarchy. Figure 10 shows the arrangement of the data in the cache lines.

5.2 Parallel Processing of Multiple Bounding Boxes

To process multiple bounding-sphere intersection tests simultaneously, the information from three bounding-spheres must be packed into a single data structure. For the minimum amount of realtime data-shuffling, the data in the "packed" data-structure is arranged and aligned in memory for quick loading with Streaming SIMD Extensions. This pre-computation phase requires that these groups of three-bounding spheres for simultaneous computation must be known before runtime. Assuming some sort of test-queuing scheme is in place to make predictable data access, this means that there are restrictions as to the actual bounding-spheres that can be put onto the queue. In particular, only these pre-selected and pre-computed groups of three bounding-spheres can be added to the queue, otherwise the congregate data-structure would need to be built on the fly with three arbitrary bounding-spheres. This realtime construction of the data not only would require a great deal of memory copying from the original bounding-sphere data structures to the congregate data-structure, but would require three arbitrary bounding-spheres' data in an unpredictable, and therefore unprefetchable, fashion.

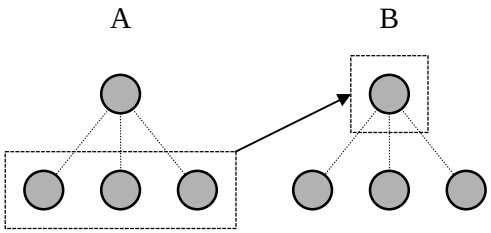
To resolve this dilemma, the observation was made that sibling bounding-spheres always require comparison to the same bounding-sphere. In particular, say that bounding-spheres `A` and `B` were determined to be intersecting. Assuming that `A` is larger than `B`, `A.Child1` and `A.Child2` must both be compared to `B`. Whenever a collision occurs, all siblings of the larger bounding-box must be compared to the smaller bounding-box. There is no reason to restrict the hierarchy to being a binary tree, so building a 3-child, or terciary tree would create groups of three bounding-spheres that are always compared to the same bounding-boxes simultaneously. Because this is known before runtime, these groups of three sibling bounding-spheres could be organized into the congregate data-structures for fast parallel computation.

The second observation was made that comparing a bounding-sphere against the *children* of a second bounding-sphere is more accurate than comparing against the second bounding-sphere itself. For example, comparing A.Child[1..n] with B is more accurate than simply comparing A and B. With this

realization, it was possible to construct an entirely new type of intersection test that is based upon simple bounding-spheres.

5.3 The Three Spheres Method

The ideal bounding-volume would contain no "extraneous" space, or space that is not actually used by the data being bounded. But, as in most cases, the ideal volume is virtually impossible to model for a general data-set (meaning unpredictably shaped objects). Therefore, the next best thing would be to have a volume that closely fits the shape of the object in such a way that *most* of the space within the bounding-volume is also contained within the data. Restated, this means that there is little space in the bounding-volume that is not contained within the data. This desire describes the hope that when two bounding-volumes are detected to be intersecting, the data that is being bounded actually intersects.

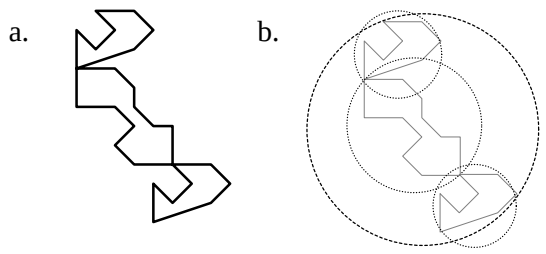


When testing for intersection between A and B, the root bounding-spheres are not compared directly. Instead, the first siblings of the bounding-sphere hierarchy of A are simultaneously compared with the root of the bounding-sphere hierarchy of B.

Traditionally, a single simple volume (such as a sphere or rectangular box) is used in the hopes that it will exhibit these characteristics for a variety of objects. A new approach would be to use the union of multiple volumes to approximate the data-space. This is the foundation of the Three-Spheres Method (see Figure 7).

Figure 7: Methods for obtaining bounding volumes

In this new algorithm, the union of three spheres is used as a bounding volume. Using Streaming SIMD Extensions technology, this new volume can be tested against a single sphere extremely quickly. Because multiple bounding volumes are used, there exists the potential for increased precision (due to less extraneous bounded space). Finally, this grouping of bounding-spheres for simultaneous testing occurs automatically with the generation of terciary trees, where the three sibling bounding-spheres are simply grouped together (Figure 8).



Part a.) of the figure shows an object to be bounded, and Part b.) shows the same object bounded by both a traditional bounding-sphere (dashed line) and the Three-Spheres Method (dotted line). Note the lessened amount of extraneous space bounded by the three spheres in place of a single sphere.

Figure 8: Testing for intersection with Bounding Spheres

6 Conclusion

The Streaming SIMD Extensions provides performance speedups when applied to several of the commonly used collision detection algorithms. Applying the Streaming SIMD Extensions to the Oriented Bounding Box algorithm can provide performance gains. Future Work

In the process of analyzing, coding, and optimizing the OBB algorithm for Streaming SIMD Extensions, a number of observations were made for possible future work:

1) Homogeneous bounding-box representation
2) Bounding-box size sorting
3) Assumption of intersection

6.1 Homogeneous Bounding-Box Representation

The intersection function of the current implementation of the OBB algorithm compares two bounding-boxes, A and B. A is considered to be located around the origin of some local coordinate system, while B is given as a translation vector and rotation matrix with respect to A's coordinate system. If the application is based upon homogenous matrices, this requires recovering the three-dimensional rotation matrix and translation vector from the existing homogenous transformation matrix in order to detect collisions – an expensive and restricting operation. In order to avoid this, the algorithm could be extended to support four-dimensional bounding boxes, and all comparisons could occur with homogenous coordinates, requiring no translation from the application's native coordinate system. In addition, this would allow for additional transformation possibilities, such as shearing and scaling, without recalculation of bounding-boxes. Preliminary results indicate that this could be done, increasing the number of conditional equations from fifteen to seventy-two. If this were done, the full potential of the Streaming SIMD Extensions could be exercised by using all four operands in the SIMD instructions.

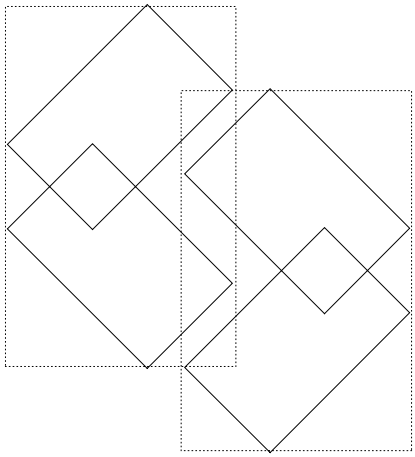
6.2 Bounding-Box Size Sorting

From the analysis that was carried out, it was realized that differing bounding-box ratios alter the probability that a given condition will "catch" a non-intersection. Because early returning from the intersection function occurs based on these conditional equations, different size ratios will alter the probability of branching. Because the penalty for mispredicting a branch is quite severe, it would be advantageous to minimize mispredicted branches. This could be done by grouping together like-sized bounding-box ratios so the Branch Prediction Unit of the Pentium® III processor will have a higher chance of correctly predicting branches based on the conditional equations.

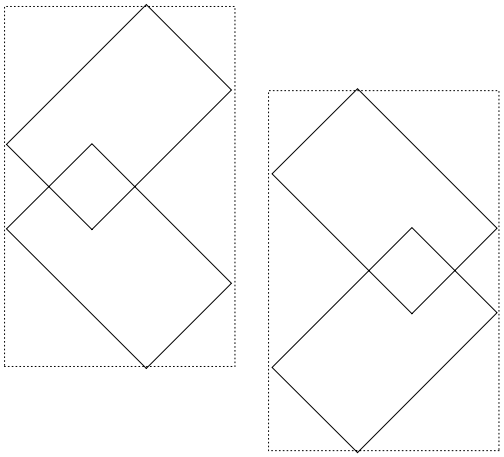
6.3 Assumption of Intersection

The most costly path through the intersection function of this study occurs when all fifteen conditional equations (or all seventy-two in the homogenous case) must be evaluated. This path is traversed under two conditions: the bounding-boxes do intersect, the bounding-boxes do not intersect but are not detected by the two classes. The statistics computed with the universe simulator show that greater than 96% of all non-intersecting bounding-boxes over the entire worst/best case range are detected within Class I and II. This means that only 4% of the non-intersecting bounding-boxes, at best a mere (0.4*0.80)=3.2% of the total number of tests, will be determined to be non-intersecting due to Class III. In addition, this final set of conditional equations must be evaluated every time bounding-boxes are in

fact intersecting – up to 20% of the total number of tests. As a result, there is an (20/23.2)=86% chance that evaluating Class III conditional equations won't "catch" non-intersection, and therefore is unnecessary.



Normally, only when parent bounding boxes (hatched lines) intersect are the child bounding boxes compared (solid lines). If the parent bounding boxes do not intersect, then by definition it is impossible for child bounding boxes to intersect.



If the child bounding boxes of two non-intersecting parent bounding boxes were tested for intersection regardless, they would still be determined as non-intersecting. As a result, "assuring" that two parent bounding boxes do intersect has no adverse effect other than potentially performing extra tests. If it is cheaper to perform the extra tests than to positively determine non-intersection, then the tradeoff may be acceptable.

To exploit this knowledge, it is required to look at the bounding-box structure. Because bounding-boxes are organized into a hierarchy, incorrectly stating that an intersection occurred between two bounding-boxes is always corrected when the child bounding-boxes are compared (because if the parent bounding-boxes do not intersect, the child bounding-boxes cannot possibly intersect). As a result, it would be possible just to assume that any non-leaf bounding-boxes that aren't determined to be non-intersecting by Class I and II actually intersect, while doing a full collision test only between leaf bounding-box nodes.

If two bounding-boxes would have been determined to be non-intersection by the last class of conditions, but was instead incorrectly determined to be intersecting due to those conditions being skipped, chances are that the intersection tests of the children bounding-boxes will detect this within the first two classes. This would in turn avoid computing the entire set of fifteen conditional equations at all levels but the leaf nodes at the cost of doing extra "abridged" intersection tests (using only Class I and II). If the current study assumed the existence of an Assumption of Intersection system, the average test time (for a particular environment) would be reduced. In addition, this reduces the number of possible branches, possibly having the effect of decreasing branch-misprediction. But, this would require identifying nodes that have no children and computing the last class appropriately. Because this

directly depends upon the depth of the hierarchy tree as well as a number of application specific factors, this improvement was not considered in the final estimate.

7 Additional Data

The charts shown in Figure 9 display the probability that a certain set of conditional equations will be the first to determine that two non-intersecting bounding boxes are, in fact, non-intersecting for a given bounding box size ratio. For instance, non-intersecting bounding boxes of similar sizes in a universe whose bounding boxes have a 20% chance of intersecting are determined to be non-intersecting within Class I 88 out of 100 times, within Class II 8 out of 100, and within Class III 4 out of 100. The charts also show the probability of non-intersection being determined by the Class I and Class II equations combined is (88 + 8) = 96 out of 100.

Figure 9: Probability charts for which equation class will determine non-intersection

Figure 10 shows the arrangement of bounding box data within two cache lines (32 bytes to a line, 4 bytes per single-precision floating-point number) for a) the Oriented Bounding Box method, and b) the Bounding Sphere method. The grayed out areas indicate unused memory reserved for storing hierarchy relationship information.

a)

$\mathbf{T}_1$	$\mathbf{T}_2$	$\mathbf{T}_3$		$\mathbf{b}_1$	$\mathbf{b}_2$	$\mathbf{b}_3$	$\mathbf{R}_{11}$
$\mathbf{R}_{21}$	$\mathbf{R}_{12}$	$\mathbf{R}_{31}$	$\mathbf{R}_{22}$	$\mathbf{R}_{32}$	$\mathbf{R}_{13}$	$\mathbf{R}_{23}$	$\mathbf{R}_{33}$

b)

$\mathbf{T1}_1$	$\mathbf{T2}_1$	$\mathbf{T3}_1$		$\mathbf{T1}_2$	$\mathbf{T2}_2$	$\mathbf{T3}_2$	
$\mathbf{T1}_3$	$\mathbf{T2}_3$	$\mathbf{T3}_3$		**R1**	**R2**	**R3**	

Figure 10: Data arrangement in cache lines